Mohamed Boudiaf
University
-M'sila-

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila

Faculty of Mathematics
and Computer Science

Computer Science
Department

OPERATING SYSTEMS II

CHAPTER II
PROCESS SYNCHRONIZATION

MUTUAL EXCLSION

3rd Year, Computer Science – SI

Dr. Hanene CHERAIT

2025-2026

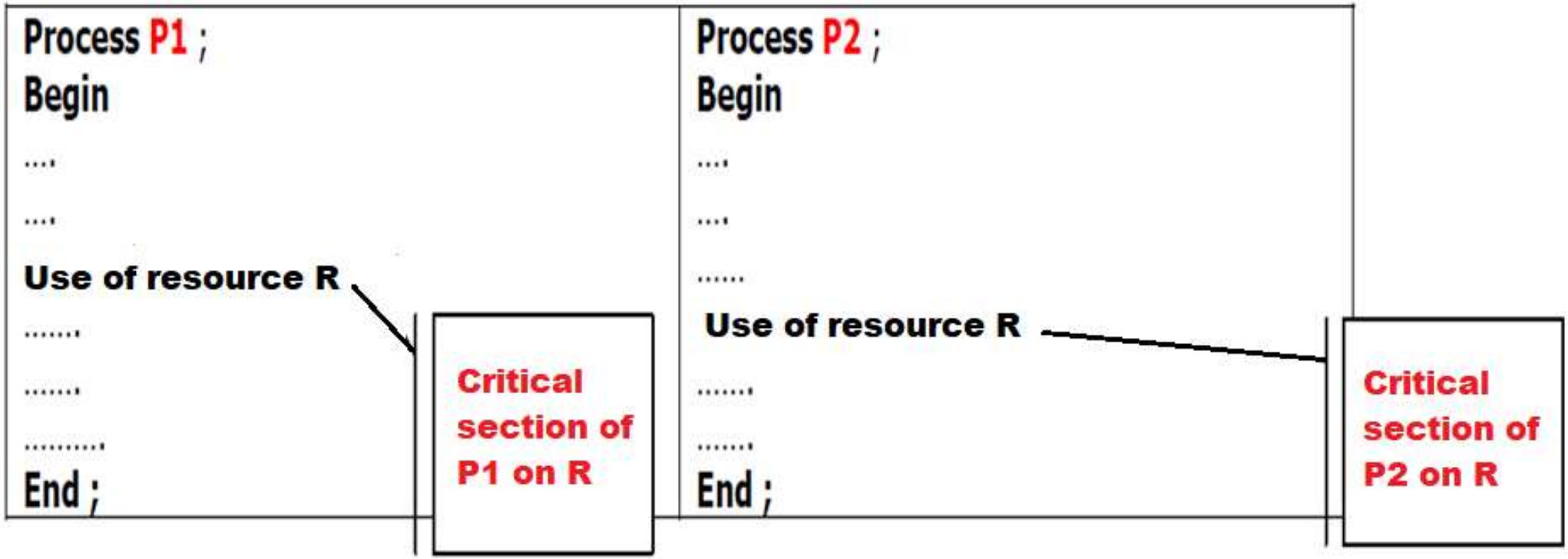
INTRODUCTION

- Several processes run in a multitasking operating system, either **pseudo-parallel** or **parallel**, using **time-sharing**.
- They **share resources**.
- Sharing objects without proper precautions can lead to **inconsistent results**.
- The solution to this problem is called **process synchronization**.

MUTUAL EXCLUSION PROBLEM

- To prevent any incorrect use of a critical resource, the critical section in different processes must **never execute simultaneously**.
- The critical sections of each process must execute in **mutual exclusion.**
- Make the sequence of instructions composing the critical section **indivisible**.

MUTUAL EXCLUSION PROBLEM



MUTUAL EXCLUSION PROBLEM

- The general principle of a solution ensures that the simultaneous execution of multiple processes **does not lead to unpredictable results.**
- Before executing a critical section (CS), a process must ensure that **no other process is currently executing a CS of the same set.**
- Otherwise, it **must not proceed** until the other process has finished its CS.

MUTUAL EXCLUSION PROBLEM

- To solve the mutual exclusion problem, four conditions must be satisfied (Dijkstra's properties):

Condition 1: Two processes can **never access their critical sections simultaneously.**
Condition 2: If no process is in the critical section and a process wants to enter its critical section, it **must be able to access it within a finite time.**
Condition 3: No process executing **outside its critical section** should block other processes.
Condition 4: All processes must **share the same solution.**

PROCESS MODEL

```
Process P1 ;
Begin
.....
```

Entry protocol

```
....
Use of resource R
```

Exit protocol

```
.....
.....
.....
End ;
```

PROCESS MODEL

```
Process P1 ;
Begin
.....
```

```
....
Use of resource R
```

```
.....
.....
.....
End ;
```

A set of elementary operations executed by P1 to ensure its exclusive entry into the critical section.

Entry protocol

Exit protocol

A set of elementary operations executed by P1 to ensure the entry of another process into the critical section

PROCESS MODEL

Shared context: variables shared by all processes

```
Process Pi;
Begin
    Repeat
        :::: // instructions outside the critical section
        EntryProtocol(); //
        CSi; // critical section of process Pi
        ExitProtocol(); //
        :::: // instructions in the remainder of the process
    Until false;
End;
```

SOLUTION FOR MUTUAL EXCLUSION

- The solution for mutual exclusion can be implemented either in:
  - Software (algorithmic)
  - Hardware

ALGORITHMIC SOLUTIONS FOR MUTUAL EXCLUSION

ALGORITHMIC SOLUTIONS FOR MUTUAL EXCLUSION / FIRST SOLUTION

```
Shared context: Turn: 0..1; //Turn initialized to 0 or 1

Process Pi;

Begin

   Repeat

   ::::

   While Turn ≠ i do;

   CSi; // critical section of process Pi

   Turn := j;

   ::::

   Until false;

End;
```

Shared context: Turn: 0..1; //Turn initialized to 0 or 1

Process P0;	Process P1;
Begin	Begin
Repeat	Repeat
::::	::::
While Turn ≠ 0 do;	While Turn ≠ 1 do;
CS; // critical section of process Pi	CS; // critical section of process Pi
Turn := 1;	Turn := 0;
::::	::::
Until false;	Until false;
End;	End;

This solution satisfies the first condition

This solution does not satisfy the second condition: a process waits for another that does not want to enter its critical section.

ALGORITHMIC SOLUTIONS FOR MUTUAL EXCLUSION / SECOND SOLUTION

```
Shared context: Flag: array [0,1] of boolean; // initialized to false

Process Pi;

Begin

   Repeat

   ::::

   While Flag[j] do;

   Flag[i] := true;

   CSi; // critical section of process Pi

   Flag[i] := false;

   ::::

   Until false;

End;
```

The process Pi that wants to enter its critical section sets its flag to true

```
Shared context: Flag: array [0,1] of boolean; // initialized to false
```

```
Process P0;
Begin
   Repeat
   ::::
   While Flag[1] do;
   Flag[0] := true;
   CS; // critical section of process Pi
   Flag[0] := false;
   ::::
   Until false;
End;
```

```
Process P1;
Begin
   Repeat
   ::::
   While Flag[0] do;
   Flag[1] := true;
   CSi; // critical section of process Pi
   Flag[1] := false;
   ::::
   Until false;
End;
```

This solution is incorrect because it violates the first condition: two processes can end up simultaneously in their critical sections

ALGORITHMIC SOLUTIONS FOR MUTUAL EXCLUSION / THIRD SOLUTION

```
Shared context: Flag: array [0,1] of boolean; // initialized to false

Process Pi;

Begin

    Repeat

    ::::

    Flag[i] := true;

    While Flag[j] do;

    CSi; // critical section of process Pi

    Flag[i] := false;

    ::::

    Until false;

End;
```

Shared context: Flag: array [0,1] of boolean; // initialized to false

Process P0;	Process P1;
Begin	Begin
Repeat	Repeat
::::	::::
Flag[0] := true;	Flag[i=1] := true;
While Flag[1] do;	While Flag[0] do;
CSi; // critical section of process Pi	CSi; // critical section of process Pi
Flag[0] := false;	Flag[1] := false;
::::	::::
Until false;	Until false;
End;	End;

Both processes can get blocked if they both execute the assignment and then the 'while' loop, which means the second condition is not guaranteed

PETERSON'S SOLUTION

Shared context: Flag: array [0,1] of boolean; // initialized to false

Turn: 0..1; //Turn initialized to 0 or 1

```
Process P0;
Begin
   Repeat
   ::::
   Flag[0] := true;
           Turn:=1;
   While (Flag[1] and Turn=1) do;
   CS; // critical section of process Pi
   Flag[0] := false;
   ::::
   Until false;
End;
```

```
Process P1;
Begin
   Repeat
   ::::
   Flag[1] := true;
   Turn:=0;
   While (Flag[0] and Turn=0) do;
   CS; // critical section of process Pi
   Flag[1] := false;
   ::::
   Until false;
End;
```

Correct solution because all four of Dijkstra's properties are satisfied

HARDWARE SOLUTIONS FOR MUTUAL EXCLUSION

DISABLE INTERRUPTS

- **Allow a process to disable interrupts while a shared variable is being modified during the execution of its critical section.**
- Clock interrupts are not possible, so the process can retain control of the CPU.
- **Bad solution:**
  - It is risky to give a user process the power to disable interrupts.
  - In a multiprocessor system, disabling interrupts affects only one CPU.
    - Other processes running on the other CPUs can still access the shared memory.

SPECIALIZED MACHINE INSTRUCTIONS

TAS (TEST AND SET)

- A primitive that consists of a few instructions designed to execute in an indivisible or atomic manner.
  - If two processes execute these instructions simultaneously, one of them will be put on hold.

```
Function TAS(var flag: boolean): boolean;
Begin
    TAS := flag;
    flag := true;
End;
```

```
Common context: Lock: Boolean; // initialized to false

Process Pi;

Begin

   Repeat

   ::::

   While TAS(Lock) do; // entry protocol

   CRITICAL SECTION;

   Lock := false; // exit protocol

   ::::

   Until false;

End;
```

```
Function TAS(var flag: boolean): boolean;

Begin

   TAS := flag;

   flag := true;

End;
```

SPECIALIZED MACHINE INSTRUCTIONS

EXCHANGE (SWAP)

- A primitive consisting of a few instructions that execute in an indivisible or atomic manner.
  - If two processes execute these instructions simultaneously, one of them will be put on hold.

```
Procedure SWAP(var flag1, flag2: boolean);
Var tmp: boolean;
Begin
    tmp := flag1;
    flag1 := flag2;
    flag2 := tmp;
End;
```

```
Common context: B: boolean; // initialized to false
Process Pi;
Begin
   var A: boolean
   Repeat
      ::::
      A:=true; // entry protocol
      Repeat SWAP(B,A) Until A=false; // entry protocol
      CRITICAL SECTION;
      B:= false; // exit protocol
      ::::
   Until false;
End;
```

```
Procedure SWAP(var flag1, flag2: boolean);
Var tmp: boolean;
Begin
   tmp := flag1;
   flag1 := flag2;
   flag2 := tmp;
End;
```

WHY USE SWAP OVER TAS?

- **More flexible for multiple flags or complex locks**
  - TAS works best for **simple binary locks** (locked/unlocked).
  - SWAP can be used for more **generalized synchronization** schemes
    - **ticket locks** or multiple state variables
    - it can exchange any value atomically, not just set a single bit.
- **Fairness in some designs**
  - TAS often leads to **starvation**
    - a process might keep losing because others repeatedly acquire the lock.
  - With SWAP, schemes can be designed where each process's "local value" or ticket helps enforce **first-come-first-served** behavior.

SWAP VS TAS

- **Atomic value exchange**
  - SWAP allows to **atomically get the previous value of the lock**, which can be useful for certain algorithms.
  - TAS only tells “was it 0 before I set it?”
    - Cannot replace it with something meaningful at the same time.

- **More general-purpose building block**
  - Some complex synchronization primitives (like **queues for locks**) are easier to implement with SWAP than TAS
    - can atomically swap pointers or more than one bit of information.

If we need a basic spinlock, TAS is usually sufficient. SWAP shines when we want to implement more sophisticated locking mechanisms.

THE BUSY-WAITING PROBLEM

- The main disadvantage of mutual exclusion solutions is **busy waiting**.
- Even if a process does nothing (i.e., it waits for the critical section), it monopolizes the CPU!
- Wastes CPU time
  - A process in busy waiting can remain indefinitely in an infinite loop
- Can lead to unexpected results

THE BUSY-WAITING PROBLEM

- **Priority Inversion**
- **Example:** if we have two processes
  - P1: high priority
  - P2: low priority
  - **Scheduling:** highest priority preemption
  - **Situation:**
    - If P2 is executing and is in its critical section (CS)
    - P1 arrives in the ready queue, so P2 is preempted and P1 takes the CPU
    - P1 waits because the critical resource is held by P2
    - P1 will never be preempted, and consequently, P2 cannot exit its CS → **DEADLOCK**
- **Conclusion:**
  - Busy waiting should not be allowed!

THANK YOU FOR YOUR ATTENTION